A Loosely Coupled Application Model for Grids

Fei Wu and K.W. Ng

Dept. of Computer Science and Engineering,
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{fwu, kwng}@cse.cuhk.edu.hk

Abstract. Scheduling distributed applications effectively and efficiently on Grid environments is difficult because of the dynamic and heterogeneous characteristics of the Internet. In this paper, we propose a loosely coupled application model for building distributed applications on Grids. We assume that a Grid application is composed of a group of independent modules. Each module performs either a remote service request or local processing. Different modules in such an application exchange information by explicitly described data that can be understood by both the application and the Grid environment. Each module is triggered by its input data, and finally it produces some output data. All information exchanges are completed transparently as they are carried out by the Grid management system. We call a module in such an application a loosely coupled module (LCM). A loosely coupled application can be defined by the combination of dependent or independent LCMs. By the loosely coupled application model, Grid applications can be built by employing discrete and heterogeneous resources on the Internet. The loosely coupled relationships among different LCMs can guarantee the robustness of the application. Parameters are defined in the application model so that application schedulers in the Grid environment can efficiently implement application scheduling by designing appropriate scheduling algorithms based on these parameters.

1 Introduction

The main goal of Grid computing [1] [2] [3] [4] is to effectively organize various computational resources distributed on the Internet to provide computing facilities to users as a large virtual computer. A Grid application can dynamically compose a large number of resources across the environment and implement its computations with high performance. The two most important features that distinguish a Grid from traditional distributed systems are heterogeneous resources and the dynamic network. Traditional distributed and parallel applications are hard to be scheduled on Grid environments because they presume a homogeneous and stable execution environment. The ease of development of Grid applications is a key problem to make the Grid a mature platform for general-purpose computing. While many studies on the development of Grid applications have been put forward, a common opinion is that current tools and languages are insufficient to develop effective and efficient applications for the Grid environment. Many issues must be tackled to bridge the gap between Grid applications and Grid environments, such as interoperability, adaptability,

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 1056 – 1065, 2005.

service discovery, application performance, large-scale data transfer, robustness, security, schedulability, etc. [5] [6]. Among these issues, robustness, adaptability and schedulability are deemed to be especially important because they promise the validity of Grid applications and guarantee the availability and performance of Grid environments.

In this paper, we propose a loosely coupled application model to guarantee the robustness, adaptability and schedulability of Grid applications. A loosely coupled application is composed of some independent software components and the corresponding data set that the components will process. Different components exchange information by explicitly described data in the data set. When running on a Grid, different components of such applications can be scheduled flexibly according to the runtime status of Grid resources. In this application model, the robustness of a Grid application can be guaranteed from two aspects: applications are composed in a loosely coupled style so as to reduce the effect on the whole applications when a partial error occurs; the necessary knowledge about Grid applications are known by the environment so that remedial actions can be applied to ensure applications can be executed correctly. As the modules in such loosely coupled applications are more independent than in tradition manners, adaptability and schedulability of these applications are much increased. The rest of this paper is organized as follows: Section 2 gives the definitions of loosely coupled applications and their properties; Section 3 describes the scheduling problems of loosely coupled applications; some future work will be outlined in Section 4, and finally, a conclusion of this work will be given.

2 The Loosely Coupled Application Model

2.1 Loosely Coupled Applications

A Grid application is a distributed application consisting of a number of components that runs in a Grid environment. The dependencies between different components can be an important factor that influences the performance of the application. In Grid environments, if the dependencies among components are weak, any partial failure will produce a smaller influence on the whole execution of the application. A strong dependency increases the probability of application failures and at the same time, the communication load. For distributed applications, the dependencies can be represented by data exchange. Suppose that *A* and *B* are two related modules in an application. If module *A* and module *B* know the internal information of each other, they can exchange data by data sharing or synchronized message passing. We call such relationship tightly coupled. The message passing model is a typical example of this class. If module *A* knows *B*'s interface and they communicate by asynchronous messages, we call it moderately coupled. Some implementations of distributed objects and Web services fall into this style. If modules *A* and *B* do not know each other, and their produced and required data are coordinated by a third-party unit or system, we say that the two components are loosely coupled. The messages in such a situation are called loosely coupled messages. These three kinds of relationships are shown in Figure 1.

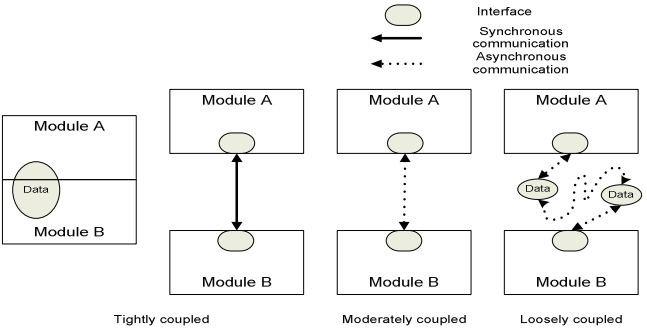

Fig. 1. Relationships between application modules. The last case shows a loosely coupled relationship between module *A* and module *B*

By such loosely coupled relationships, the influence among modules is much reduced. Partial failures won't cause the whole application to crash. Once execution resumption mechanisms are introduced, the application can complete its computation successfully even when normal errors or exceptions occur. Thus the robustness of the application can be guaranteed. The loosely coupled structure also reduces the difficulty of resource co-allocation since the resource allocation for one module affects little on the resource allocation for another module.

We define a *loosely coupled module (LCM)* as an individual module of a distributed application that can be scheduled independently onto remote or local resources. It can be either a remote service request or local processing. A LCM communicates with other parts of the application by loosely coupled messages. A *loosely coupled application (LCA)* is a distributed application that contains LCMs. How to schedule LCMs is an important part of work in the scheduling of the whole application. For convenience, in the rest of the paper, when we say scheduling of loosely coupled applications, we actually indicate scheduling of a group of LCMs.

The loosely coupled application model is part of our framework of Service-based Heterogeneous Distributed computing (SHDC) [7]. In our framework, implementing such loosely coupled applications mainly involves three aspects of work. On the application level, each component can be designed independently, and communication among these components is completed by either file exchanging or explicit messages. On the system level, tools are needed to administrate various services, schedule applications and coordinate data communication. On the SHDC framework level, services in the Internet are organized into a P2P network and distributed applications are scheduled by the cooperation of different peers [8]. Moreover, powerful description methods are required to enable an application and the system to understand each other. The descriptions shall have the ability to describe the necessary semantic contents of applications so that they can be scheduled rightly by system-level tools.

2.2 The Loosely Coupled Application Model

We define a loosely coupled application as a set *M* containing *n* modules: M_0, M_1, …, M_{n-1}. Each module M_i is composed of 5 elements: {$attribute_i$, {$inputmsg_{pi}$, …,

$inputmsg_{qi}$}, {$outputmsg_{jk}$, …, $outputmsg_{jm}$}, {$term_{pi}$, … $term_{qi}$}, L_i}. $Attribute_i$ holds the attributes of the corresponding component such as its functionality, execution requirements. The *attribute* element provides the necessary information of a component so that it can be scheduled onto proper resources. The element $inputbuf_{pi}$ holds messages received from module M_p ($0 \le p \le n-1$) but M_i hasn't processed the messages by internal computation steps. The element $outputbuf_{ik}$ holds messages that M_i wants to send to module M_k ($0 \le k \le n-1$) but the messages haven't been delivered out. The *term* element is used to monitor input messages to ensure they conform to the requirement of the corresponding modules. When any exception occurs on the input messages, the corresponding events will be issued by the *term* element so that the exceptions can be dealt with. A key function of the *term* element is to solve the non-response problems in an asynchronous system. It can satisfy the acceptable period of time of receiving a message. When the deadline reaches and the message has still not arrived, an exception event can be issued. Another element is the logic element *L* which monitors the output messages. It is used to produce communication events to transfer the output messages to other modules. It is useful especially when implementing logic controls among a group of components: when the output satisfies some conditions the data transfer can be re-directed by element L_i. This loosely coupled application model is a state-based model. Each state of module M_i contains three sets: {$inputmsg_{pi}$ …$inputmsg_{qi}$}, {$outputmsg_{ik}$…$outputmsg_{im}$}, {$term_{pi}$ … $term_{qi}$}. The *attribute* element and the logic element are not included in M_i's state since they are predefined and unchangeable. The state set Q_i contains a distinguished subset of initial states and a distinguished subset of terminal states. In an initial state every $inputmsg_{pi}$ must be empty.

The module M_i's states, except for the $outputms_{gi}$ (because in an asynchronous system, the computation will be triggered by the input, the output will not influence the state transition), comprise the accessible states of M_i. When the transition function accepts an input value of the accessible state of M_i, it produces a value of the accessible state of M_i as output in which outdated data in $inputbuf_i$ is cleared. It also produces as output at most one incident message to every other module in M. Each step processes the necessary messages waiting to be delivered to M_i and results in a state change and at most one message to be sent to every other module. When there is an input message $inputmsg_{ij}$ or an output message $outputmsg_{pq}$, we say that module j is dependent on module i, or module q is dependent on module p, denoted by *DEP(i,j)* and *DEP(p,q)* respectively.

There are five kinds of normal events in the system. One kind is a computation event, denoted *computation(i)*, representing a computation step of module M_i in which M_i's transition function is applied to its current accessible state. When a component finishes its computation, a *finishcomputation(i)* event will be created denoting that the computation result is ready to be further used. Another kind of events is a delivery event, denoted *delivery(i,j,m)*, representing the delivery of message m from module M_i to M_j. The fourth kind of events is exception events, denoted *termexception(p,i,m)*, representing that module M_i cannot receive (or accept) message *m* from module M_p according to the setting of $term_{pi}$ due to either a network exception or a computation error or a service fault. The fifth kind of events is communication events, denoted *communication (i,j,m)*, representing that module M_i will send message *m* to module M_j.

A configuration is a vector

$$C = (q_0, \ldots, q_{n-1})$$

where q_i is a state of module M_i. The states of the *outputmsg* variables in a configuration represent the messages that are in transit on the communication channels. An initial configuration is a vector $(q_0, \ldots q_{n-1})$ such that each q_i is an initial state of M_i. The behavior of a system over time is modeled as an execution, which is a sequence of configurations alternating with events. An execution is a (finite or infinite) sequence of the following form:

$$C_0, \varphi_1, C_1, \varphi_2, C_2, \varphi_3, \ldots$$

where each C_k is a configuration and each φ_k is an event. If the execution is finite then it must end in a configuration. Furthermore, several conditions must be satisfied:

1. If φ_k = *delivery(i,j,m)*, then *m* must be an element of $outputmsg_{ij}$ in C_{k-1}. The only changes in going from C_{k-1} to C_k are that *m* is added to $inputmsg_{ji}$ in C_k. In another word, a message is delivered only if it is arrived and the only change is to copy the message from the sender's outgoing message buffer to the recipient's incoming message buffer.

2. If φ_k = *computation(i)*, then the only changes in going from C_{k-1} to C_k are that M_i changes state according to it's transition function operating on M_i's accessible states in C_{k-1} and M_i will not be accepting any input messages during the computation time to ensure the computation can be rightly implemented.

3. If φ_k =*finishcomputation(i)*, then the only changes in going from C_{k-1} to C_k are that M_i changes state according to its transition function and the output messages are produced and ready for further communication. At the same time, the set of input messages specified by M_i's transition function are removed from $outputmsg_{pi}$ in C_k.

4. If φ_k = *termexception(i,p,m)*, then the only changes in going from C_{k-1} to C_k are that either a communication request is sent out to re-transfer messages *m* from module M_p (when the *termexception* event is caused by a network failure) or the state of M_p is set to it's initial state (when the *termexception* event is caused by a computation failure), and then reset the term settings relevant to module M_p in C_k.

5. If φ_k = *communication(i,j,m)*, the system doesn't change its state. Moreover, the message *m* is supposed to be still accessible after the communication events and delivery events occur to ensure the application's robustness. After the asynchronous message *m* has been sent from M_i to M_j, a delivery event will be produced.

We assume that all events are produced by a unified application controller or can be notified to the controller before the next actions. Thus all of the events in the model can be arranged into an event queue by a unified time. We use *time(q)* to denote the time that event q occurs.

The execution time of a module M_i at phase *k* is:

$$execT(i_{phaes\text{-}k}) = time(finishcomputation(i_{phaes\text{-}k})) - time(computation(i_{phaes\text{-}k}))$$

The transfer time of a message *m* from module M_i to M_j at phase *k* is:

$$transT(i,j,m_{phaes\text{-}k}) = time(delivery(i,j,m_{phaes\text{-}k})) - time(communication(\ i,j,m_{phaes\text{-}k}))$$

The execution time of the application is:

$$T = max\ \{time(delivery\ (i,j,m))\}$$

$0<=i,j<=n-1$, m is any possible message that carries the results of the computation.

The unstable network status and heterogeneous Internet services are two important characteristics of Grid environments. The robustness of an application becomes a basic requirement. The loosely coupled application model proposed here emphasizes the fault-tolerant issue in two aspects. One of the functions of the *terms* in the model can be used to solve the problems of asynchronous messages passing. With these terms, network faults and service faults can be detected, the corresponding data can be re-delivered and necessary application modules can be re-scheduled. Communication between different applications modules are implemented by buffered asynchronous message passing. Each output buffer is reserved until the data in the buffer has been dealt with by the next step in the computation successfully. This buffer mechanism may lead to some storage waste, but can guarantee computations to be executed accurately even when a network fault or a service fault occurs.

The *termexception* event is the key to detect a network or resource exception. The *termexception* events are created by the constraints of input messages. Application designers can set term elements to ensure the input messages satisfy some conditions. When a message cannot satisfy a condition, a *termexception* event will be created. Usually a term element $term_i$ is in such a form:

$$term_i: \{message\ m;\ condition\ c;\ actions\}$$

It implies that when message m cannot satisfy condition c, the following actions will be issued. Usually those actions are to create some *termexception* events. The condition c can be any conditions to restrict the messages, such as message size, message precision, etc. Moreover, one important function of the term element is to limit the arrival times of messages. Actually this functions like a timer. When an appointed message has not arrived during the prescribed time period, the corresponding events will be created to inform the system or other modules.

Suffering from the explicit message communication, applications designed in this model might meet problems for some uncertain messages. For an example, if one component in the application is defined as:

```
component A
{
…
if (condition1) then send message m to component B;
if (condition2) then send message m to component C;
…
}
```

To deal with the uncertain messages, the logic element can be introduced to implement control logics at the component-level. Each logic element L_i is in such a format:

$$L_i: \{message\ m;\ condition\ c;\ actions\}$$

in which m is a message, c is a logic expression, *actions* are usually communication events. When message m satisfies the condition c, the *actions* will be issued.

To apply the logic element to the above example (suppose components *A*, *B* and *C* are included in modules M_i, M_j and M_k respectively):

```
component A
{
…
produces message m;
…
}
L0: {m; condition1; communication(i,j,m)}
L1: {m; condition2; communication(i,k,m)}
```

By producing communication events according to conditions at runtime, the loosely coupled application model can implement complex logic controls such as branches and loops at the component-level. In this way, the modularization of each component is largely enhanced. The complexity of the design of each component is reduced and the schedulability of the whole application is improved.

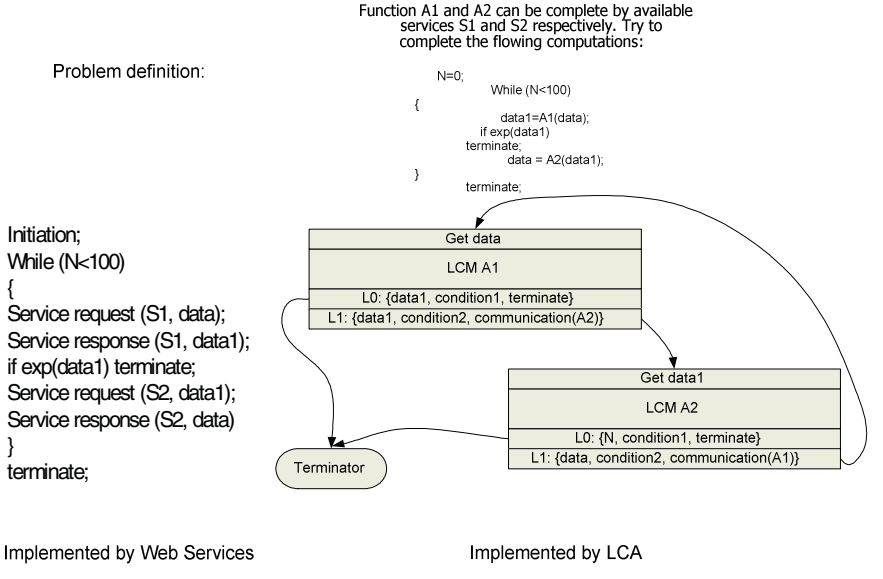

Fig. 2. An example application implemented by Web services and the loosely coupled model. The implementation by LCMG is more modularized than by Web services. It introduces only two modules and at most 200 data communications. The implementation of Web services style may cause 200 possible service requests and 400 possible data communications

This loosely coupled application model is based on services. Also there are other service-based models such as Web services and Grid services. The most important

difference between this model and other work is that modules of an application in our model are absolutely loosely coupled: each module only interacts with the outside world by reading or writing predefined format of messages; different modules do not need to know each other even when they are dependent; the global execution flow is understandable to the system so that various exceptions can be caught and dealt with; applications' robustness can be guaranteed. Distributed applications in the loosely coupled model are well modularized and each module in the applications is comparatively independent. By Web services or Grid services methods, services are called from programs, and application developers must face various exceptions caused by the network or services. But in our model, each module in an application can be designed independently, and can be developed in any programming languages, software or hardware tools and remote services. As long as the interfaces of the modules are correctly designed, the application can be scheduled by the Grid management system efficiently. Moreover, data caching is an important feature of the model. Input and output can be cached on the server side, and they can be transferred to any other server by the direction of the application scheduler dynamically. By using proper scheduling algorithms, the communication cost can be dramatically reduced, and at the same time, applications can be executed with better performance. We give an example application that implemented by Web services and LCMG respectively in Figure 2.

3 Schedule of Loosely Coupled Applications

The procedure of mapping an application onto computing resources according to some rules to implement the computation is called task scheduling. The objective of task scheduling is to order the execution of applications so that task precedence requirements are satisfied and a minimum schedule length is provided. Task scheduling is one of the most important subjects that have been extensively studied in parallel computing and distributed computing. The loosely coupled application model we proposed in our framework largely weakens the relationship between different application components, and provides an explicit structure to increase the schedulability of Grid applications. The efficiency and effectiveness of scheduling algorithms can largely influence the performance of the application. There are many scheduling algorithms based on various computing platforms. But traditional scheduling algorithms are mostly based on shared-memory systems or a cluster of workstations, they cannot be used on such heterogeneous scheduling problems. In this section, we present the definition of scheduling a loosely coupled application onto a heterogeneous distributed system.

We define a heterogeneous distributed system D as: $D = \{S, C, T, P\}$, where T is the attribute set of services; S is a finite set of services, each element S_i represents a service that can be employed by applications, for $\forall\ S_i \in S$, $S_i = \{T_i \mid T_i \subseteq T\}$; C is a communication cost matrix, for $\forall\ 1 <= i, j <= N$, $C_{ij} \in C$, C_{ij} is the communication cost between service S_i and S_j, and P is a set of dependency functions describing the dependence relationship between different services, for $\forall\ P_i \in P$, $P_i = f_i(S_{r0}, \ldots S_{rk})$, denoting that service S_i is dependent on services S_{r0}, S_{r1}, … and S_{rk}, and there is a function $f_i()$ that can be used to compute the influence on performance that services S_{r0} to S_{rk} have on S_i.

A loosely coupled distributed application DA is defined as: DA = {MODULE, DATA, T, COST}, where MODULE is a finite set of n modules, DATA is a finite set of data, T is a finite set of attributes, COST is a finite set of functions to predict the performance of modules. For $\forall$ 0<j<n+1, $MODULE_j \in$ MODULE, $MODULE_j$ = {($input_j$, $output_j$, $cost_j$, AT_j)| $input_j \subseteq$ DATA, $output_j \subseteq$ DATA, $cost_j \in$ COST, $AT_j \subseteq$ T}. In this definition, $input_j$ is the set of data module $MODULE_j$ requires. Once $input_j$ is ready, the module $MODULE_j$ can start its execution. While $output_j$ is the set of data the module $MODULE_j$ produces. The function of $cost_j$ is used to approximately evaluate the computation cost of $MODULE_j$. AT_j contains attributes of the module.

A general schedule scheme is a map from the task graph to the target system: f: DA $\rightarrow$ D $\times$ [0, ∞]. f(i) = (S_j ,t_i) means module $MODULE_i$ is scheduled onto service S_j, and its predictable start time is t_i. A module $MODULE_i$ can be scheduled onto service S_j if $AT_i \subseteq Ts_j$. The time that all modules complete execution and return results is called the Schedule Length (SL). One of the aims of a scheduling algorithm is to reduce SL to as small a value as possible. Such a task scheduling problem is shown to be NP-complete [9]. As a Grid environment may contain a huge number of resources or services, it is impossible for the scheduling algorithm to map a loosely coupled distributed application based on all possible resources that can satisfy the application's requests. How to select appropriate resources and make better scheduling schemes is an important issue for ensuring both client's and system's performance.

We can use the definitions of the loosely coupled application model to define some parameters such as adaptability and schedulability to implement various scheduling algorithms for Grid applications. For example, we give a simple definition of adaptability below. Suppose there are limited services in the environment, the number of the services is N. For any module M_i in an application that contains n modules (0<=i<=n), if there are K services that can satisfy the requirements of module M_i, we denote the adaptability of module M_i as $adaptability(M_i)$ = (K-1)/N. For those modules that can only be scheduled onto one resource, the adaptability is always 0. The parameter of adaptability can be used in scheduling algorithms for applications to achieve better performance or for the system to keep its usability. For example, we can give a simple insufficient resource first algorithm based on the adaptability of each module. This algorithm schedules modules with lower adaptability first to avoid potential resource conflicts in later computations.

```
while (there are un-scheduled modules)
{
  while (there are ready-for-scheduling modules)
  {
     select a module with lowest adaptability: Mi;
       if (there is available resources to schedule Mi)
      {
          schedule Mi;
          mark Mi as scheduled;
       }
    }
}
```

4 Conclusions

In this paper, we have introduced a loosely coupled application model which can be used to model Grid applications. Comparing to other models, this application model is powerful to model Grid applications more directly and efficiently; and at the same time it can guarantee the robustness, adaptability, and schedulability of Grid applications.

Acknowledgements

The work described in this paper was partially supported by the following grants: RGC Competitive Earmarked Research Grants (Project ID: 2150348, RGC Ref. No.: CUHK4187/03E ; Project ID: 2150414, RGC Ref. No.: CUHK4220/04E).

References

1. M. Baker, R. Buyya and D. Laforenza, "Grids and Grid Technologies for Wide-Area Distributed Computing", *Software - Practice and Experience 32*(15): 1437-1466 (2002).
2. I. Foster, and C. Kesselman, (Eds.), *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 2004.
3. I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid", *International Journal on Supercomputing Applications* 15(3):200-222, 2001.
4. I. Foster, C. Kesselman, J.M. Nick, and S. Tueche. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", *Open Grid Service Infrastructure WG*, Global Grid Forum, June 2002.
5. C. Lee and D. Talia, "Grid programming models: current tools, issues and directions", in *Grid Computing – Making the Global Infrastructure A Reality*, F. Berman, G.C. Fox, and A.J.G. Hey, (Eds.), John Wiley, 2003.
6. D. Bader, *et al*., "The Role and Requirements of Grid Programming Models", www-unix.Gridforum.org/mail_archive/ models-wg/pdf00002.pdf
7. F. Wu and K.W. Ng, "A Toolkit to Schedule Distributed Applications on Grids", *Fourth International Network Conference*, pp. 11-18, UK, 2004.
8. F. Wu and K.W. Ng, "SHDC: A Framework to Schedule Loosely Coupled Applications on Service Networks", *Grid and Cooperative Computing* - GCC 2004: Third International Conference, Wuhan, China, October 21-24, 2004.
9. R.L. Graham. "Bounds on multiprocessing anomalies." *SIAM Journal of Applied Mathematics* , 17(2): 416-429, 1969